МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ

ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

ОТЧЕТ по

лабораторной работе №4

по дисциплине «Алгоритмы и структуры данных»

Тема: Деревья

Студент гр. 8304 Самакаев Д.И.

Преподаватель Фирсов М.А.

Санкт-Петербург

2019

Вариант 10-д

Цель работы.

Изучить основные принципы работы с деревьями и лесами, и принципы их обработки.

Постановка задачи.

Формулу вида
< формула > ::= < терминал > | (< формула > < знак > < формула >)
< знак > ::= + | – | *
< терминал > ::= 0 | 1 | ... | 9 | a | b | ... | z можно представить в виде бинарного дерева («дерева-формулы») с элементами типа Elem=char согласно следующим правилам:

- формула из одного терминала представляется деревом из одной вершины с этим терминалом;

с помощью построения дерева-формулы t преобразовать заданную формулу f из инфиксной формы в префиксную (перечисление узлов t в порядке КЛП) и в постфиксную (перечисление в порядке ЛПК); преобразовать дерево-формулу t, заменяя в нем все поддеревья, соответствующие формулам (f1 * (f2 + f3)) и ((f1 + f2) * f3), на поддеревья, соответствующие формулам ((f1 * f2) + (f1 * f3)) и ((f1 * f3) + (f2 * f3)).

Описание алгоритма.

Считывается выражение, после чего по его значениям рекурсивно заполняется дерево. После этого путём изменения указателей в дереве, формула меняется на указанную в задании.

Спецификация программы.

Программа предназначена для арифметического вычисления значения выражений, представленных в постфиксной форме.

Программа написана на языке C++. Входные данные подаются в виде строк текстового файла или консольным вводом.

Описание функций.

1. bool is_brackets_correct(std::string &expression)

Определяет, правильно ли в строке expression расставлены скобки.

2. void fill_map(std::shared_ptr<Branch> temporary, size_t depth, std::map<size_t, std::string> &depth_root_map)

Заполняет словарь по принципу глубина - корень.

3. bool create_node(std::string& expression, int &i, Node* &element)

4. void replace_with_associative(std::string s)

Меняет дерево в соотетствии с заданием.

5. void addRoots(std::shared_ptr<Branch> temp, std::string s)

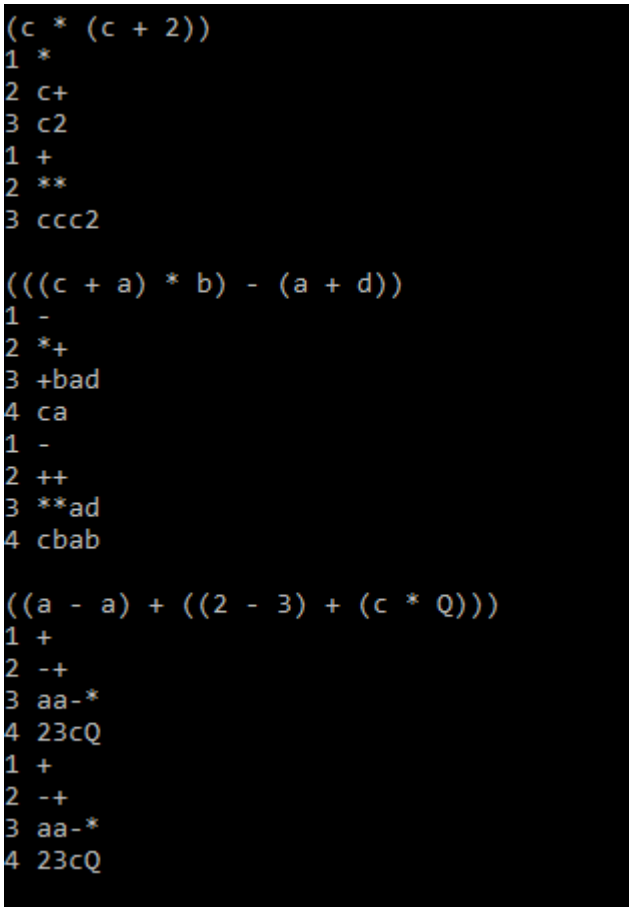
Рекурсивно заполняет дерево.

Вывод.

Была реализована программа, позволяющая строить бинарные деревья по заданной форме, а так же упрощать их.

Приложение.

1)Тестирование.



(c * (c + 2))	1 * 2 c+ 3 c2	1 * 2 c+ 3 c2
(((c + a) * b) - (a + d))	1 - 2 *+ 3 +bad 4 ca	1 - 2 ++ 3 **ad 4 cbab
((a - a) + ((2 - 3) + (c * Q)))	-1 + 2 -+ 3 aa-* 4 23cQ	-1 + 2 -+ 3 aa-* 4 23cQ

2)Исходный код.

Lab4.cpp:

```
#include "lab.h"


void delete_space_symbols(std::string & expression) {
	expression.erase(std::remove_if(expression.begin(), expression.end(), &isspace),
expression.end());
}

bool is_brackets_correct(std::string& expression) {

	int brackets_cnt = 0;

	for (size_t i = 0; i < expression.length(); i++) {
		if (brackets_cnt < 0)
			return false;
		else {
			if (expression[i] == '(')
				brackets_cnt++;
			else if (expression[i] == ')')
				brackets_cnt--;
			else continue;
		}
	}
	if (brackets_cnt == 0)
		return true;
	else return false;
}


void console_input() {
	BinTree tree;

	std::cout << "Please, enter the expression" << std::endl;
	std::string expression;

	getline(std::cin, expression);
	if (is_brackets_correct(expression)) {
		delete_space_symbols(expression);

		tree.replace_with_associative(expression);
	}
	else std::cout << "check if the brackets are correct" << std::endl;
}

void file_input(char* argv) {

	std::ifstream file;
	std::string testfile = argv;

	file.open(testfile);
	if (!file.is_open()) {
		std::cout << "Error! File isn't open" << std::endl;
		return;
	}

	std::string expression;

	while (!file.eof()) {
		BinTree tree;
```

```
		getline(file, expression);

		std::cout << expression << std::endl;
		if (is_brackets_correct(expression)) {

			delete_space_symbols(expression);

			tree.replace_with_associative(expression);

			std::cout << std::endl;
		}
		else std::cout << "check if the brackets are correct" << std::endl;
	}

}

int main(int argc, char** argv) {

	if (argc == 1) {
		console_input();
	}
	else file_input(argv[1]);

}
```

Lab.h:

```
#pragma once
#include <iostream>
#include <fstream>
#include <string>
#include <cctype>
#include <map>
#include <algorithm>

struct Branch {
	Branch() = default;

	std::string root = "0";
	std::shared_ptr<Branch> left = nullptr;
	std::shared_ptr<Branch> right = nullptr;
};


class BinTree {
public:
	BinTree() {
		head = std::make_shared<Branch>();
	}

	void addRoots(std::shared_ptr<Branch> temp, std::string s) {

		size_t bracket_cnter = 0;
		size_t i = 0;

		if (isalpha(s[0]) || isdigit(s[0])) {
			temp->root = s[i];
			return;
		}

		for (size_t i = 0; i < s.length(); i++) {
			if (s[i] == '(') {
				bracket_cnter++;
			}
			else if (s[i] == ')') {
```

```
                        bracket_cnter--;
                }

                if ((s[i] == '+' || s[i] == '-' || s[i] == '*') && bracket_cnter == 1) {

                        temp->left = std::make_shared<Branch>();
                        temp->right = std::make_shared<Branch>();

                        temp->root = s[i];

                        addRoots(temp->left, s.substr(1, i - 1));

                        addRoots(temp->right, s.substr(i + 1, s.length() - i - 1));

                        return;
                }
        }
    }

    void fill_map(std::shared_ptr<Branch> temporary, size_t depth, std::map<size_t,
std::string> &depth_root_map) {

        depth++;

        if (!temporary->left || !temporary->right) {
                if (depth_root_map.find(depth) != depth_root_map.end())
                        depth_root_map[depth] += temporary->root;
                else depth_root_map.insert(make_pair(depth, temporary->root));
                return;
        }

        fill_map(temporary->left, depth, depth_root_map);

        fill_map(temporary->right, depth, depth_root_map);

        if (depth_root_map.find(depth) != depth_root_map.end())
                depth_root_map[depth] += temporary->root;
        else depth_root_map.insert(make_pair(depth, temporary->root));
    }

    void print_tree(std::map<size_t, std::string>& depth_root_map) {
        for (auto it = depth_root_map.begin(); it != depth_root_map.end(); it++)
                std::cout << it->first << " " << it->second << std::endl;
    }

    void replace_with_associative(std::string s) {
        addRoots(head, s);

        fill_map(head, depth, depth_root_map);
        print_tree(depth_root_map);

        change_tree(head);

        depth_root_map.clear();

        fill_map(head, depth, depth_root_map);
        print_tree(depth_root_map);

    }

    void change_tree(std::shared_ptr<Branch> temporary) {
        if (!temporary->left || !temporary->right) {
                return;
        }
```

```
            if (temporary->root == "*") {
                if (temporary->left->root == "+") {
                    Branch buffer = *temporary->left->right;
                    temporary->left->right = temporary->right;
                    temporary->left->root = "*";
                    temporary->root = "+";
                    temporary->right = std::make_shared<Branch>();
                    temporary->right->left = std::make_shared<Branch>();
                    temporary->right->right = std::make_shared<Branch>();
                    temporary->right->root = "*";
                    *temporary->right->left = buffer;
                    temporary->right->right = temporary->left->right;
                }
                else if (temporary->right->root == "+") {
                    Branch buffer = *temporary->right->left;
                    temporary->right->left = temporary->left;
                    temporary->right->root = "*";
                    temporary->root = "+";
                    temporary->left = std::make_shared<Branch>();
                    temporary->left->right = std::make_shared<Branch>();
                    temporary->left->left = std::make_shared<Branch>();
                    temporary->left->root = "*";
                    *temporary->left->right = buffer;
                    temporary->left->left = temporary->right->left;
                }
            }

            change_tree(temporary->left);
            change_tree(temporary->right);
        }

    private:
        std::shared_ptr<Branch> head;
        std::map<size_t, std::string> depth_root_map;
        size_t depth = 0;
};
```